Executable/Translatable UML in Computing Education

Shayne Flint **Henry Gardner** **Clive Boughton**

Department of Computer Science
Computer Science and Information Technology Building
The Australian National University
Canberra ACT 0200
Australia
Email: shayne.flint@anu.edu.au

Abstract

The Unified Modelling Language (UML) is a diagrammatic notation widely used in the computing industry and often taught in universities as a way to represent software requirements specifications and design descriptions. In this paper we identify a number of problems associated with teaching UML and how we have minimised their impact by making use of the Executable/Translatable UML ($\frac{X}{T}UML$). We describe two case studies that demonstrate the benefits we have gained by using $\frac{X}{T}UML$ in our undergraduate and graduate teaching programs.

Keywords: UML, Executable UML, Education, Shlaer-Mellor, Virtual Reality

1 Introduction

The Unified Modelling Language (UML) is a diagrammatic notation that is widely used in the computing industry to specify, visualise, and document models of software structure and behavior. Because of its wide acceptance by the computing industry, few students of computing-related disciplines survive to the end of their programs without some contact with UML (OMG 2003). In particular, UML is often taught as a way to represent software requirements specifications and design descriptions.

Because UML is only a notation for representing software models, there is a need to teach it together with an appropriate analysis and design method. Such methods are often based on the work of Booch and Rumbaugh (Booch 1994, Rumbaugh, Blaha, Premerlani, Eddy & Lorenson 1990) and might be called *elaborative* in that they begin during analysis by creating object-oriented models of software at a high level of abstraction, omitting design and implementation details. During design phases these models are *elaborated* (or refined) to include design and implementation information. Eventually the models become specifications for code. UML is used to represent models produced at all the stages of this process.

We see three problems with this situation.

1. Because of its wide range of applicability and because of its history (as a *unification* of three different notations) UML is very large and, in places, ambiguous. We believe that its size and complexity is beyond what should be taught to undergraduate students.
2. We believe that popular elaborative approaches to development are at odds with important software engineering principles that we wish to teach to our students. In particular, elaborative methods encourage a blurring of analysis and design activities and work products. This conflicts with our ability to teach and demonstrate the importance of separating concerns. In addition, methods that encourage the development of partial models conflict with industry accepted indicators of good specifications including correctness, lack of ambiguity, completeness, consistency, verifiability and maintainability (IEEE 1998).
3. Students want to use the lastest implementation technology including Java, PHP, .Net, and Python. Unfortunately, many of these technologies offer poor support for the software engineering principles we wish to teach. The result is that student design models are often influenced by idiosyncrasies of the implementation technologies being used. This further complicates the teaching of software engineering principles.

In this paper we introduce Executable/Translatable UML ($\frac{X}{T}UML$) (Mellor & Balcer 2002) and show how it has been used in our undergraduate and graduate teaching program to deal with each of the problems outlined above. We follow this with a description of two, very different, case studies. The first is a simple web based application implemented using PHP. The second is a simple Virtual Reality application implemented using Java.

2 Executable and Translatable UML: Key Concepts

2.1 $\frac{X}{T}UML$ is a translative method of software development

Object oriented software development methods can generally be classified as either *elaborative* or *translative*.

Elaborative methods of software development are the most popular at the present time and are taught in most software analysis and design courses. They are based on the belief that object orientation can be used to smooth the transition from analysis to design and implementation. An *elaborative* approach begins during analysis by creating object-oriented models of software at a high level of abstraction, omitting design and implementation details. During design phases these models are *elaborated* (or refined) to include design and implementation information thus blurring the distinction between requirements specification and design descriptions. Eventually the models become specifications for code.

Translative methods of software development are based on the idea of maintaining a clear separation of concerns throughout the entire software lifecycle. These separated concerns include aspects associated with requirements specifications, software architecture, and implementation.

Models produced during analysis activities specify the required data, state and behavior of separate aspects of a software system independent of implementation. Information from these models is then *translated* and *woven* together with *separately developed software architecture and implementation models* to form code and other software engineering artifacts.

The translative approach to software development was pioneered by Sally Shlaer and Stephen Mellor in the 80's leading to the development of the *Shlaer/Mellor method* (Shlaer & Mellor 1992). During recent years the Shlaer/Mellor method has been refined and updated to make use of a well defined subset of UML for representing models which can be executed for verification and simulation purposes. This method is now called Executable/Translatable UML ($_{T}^{X}UML$) and is described in a number of recent books (Mellor & Balcer 2002, Starr 2002, Starr 2001). The method is also supported by at least two commercial software engineering tools (Project Technology 2003, Kennedy Carter 2003). In our teaching program we use (Mellor & Balcer 2002) as the text book and (Project Technology 2003) in our laboratories.

2.2 Domains

$_{T}^{X}UML$ supports the separate modeling of the various subject matters that comprise a software system. An $_{T}^{X}UML$ model comprises a set of *domain* models that each capture the concepts of an autonomous subject or aspect of a system such as the application itself, user and other interfaces, databases, security and networking. While some domains may appear to be subsystems, most are not. Most domains specify a particular aspect of a system which may touch all parts of the final software. An example of such a domain is security. A security domain may specify security-related concepts such as users, roles, passwords, and privileges, how the concepts relate to each other, and how they respond to various stimuli. Security may affect many parts of an operational software system but nonetheless is modeled and verified independently of any other subject matter. During model translation, subject matter of the application domain, security domain and other domains will be systematically *translated* and *woven* together to form a working software system.

Domains can be categorised as *application*, *intermediate abstractions*, or *implementation.* Application domains deal with software requirements and contain no design or implementation concerns. Implementation domains, on the other hand, deal with design and technology concerns such as software architecture, programming languages and communications. A working software system will be constructed from the concepts defined in implementation domains. Intermediate abstractions are domains used to decouple an application requiring generic services, such as a database, from the specific technology that provides the service. Some of these domains may be well understood or they may already exist. These *realised* domains are not modelled using $_{T}^{X}UML$ and usually represent implementation technologies and external systems with which the subject software must interact, including off-the-shelf or legacy software.

Figure 1 depicts a domain chart showing the software aspects of a simple system. The implementation domains represent various aspects of the popular open source LAMP architecture - Linux (Operating system), Apache (web server), PHP (programming language) and MySQL (database). The Persistence Domain is an intermediate abstraction that decouples application domains from implementation technology (MySQL). The Bookmark and Security domains model application requirements. This example will be described in more detail in the first case study below.

2.3 Executable domain models

An important feature of $_{T}^{X}UML$ is that domain models are complete executable models of a given subject matter. This means that domains dealing with requirements are executable specifications which allow for verification of required functionality early in the software lifecycle before any design or implementation technologies are even considered. To facilitate the construction of executable models, $_{T}^{X}UML$ domains are represented using a well defined and integrated subset of UML comprising class diagrams (no operations), state charts, collaboration diagrams and sequence diagrams together with the Object Constraint Language (OCL) and an implementation of the UML action semantics.

Note that while $_{T}^{X}UML$ domain models are represented using a small subset of UML and include an asynchronous view of behavior, they place no restrictions on how domain subject matter is translated. For example, domain models can be translated into object-oriented or structured designs, or directly into object oriented, structured or unstructured programming languages. An asynchronous specification can be implemented synchronously.

2.4 Bridges

Application domains are usually modeled independently of issues such as security, user interaction and persistence. They rely on *bridges* to other domains to deal with such issues. These bridges are shown as dashed arrows on the domain chart and represent the assumptions made by a client domain and a set of corresponding requirements that are placed on some other relevant server domain.

For example, the arrow between the Bookmark and Graphical User Interface (GUI) domains depicted in Figure 1 indicates that the Bookmark domain assumes that some other (anonymous) domain will provide a user interface and that in this case the bridge has placed corresponding requirements on the GUI domain.

The consequences of this bridging concept include the ability to replace or supplement server domains without changing any client domains. For example, the GUI domain depicted in Figure 1 could be replaced, at the level of specification rather than design or implementation, with a domain that models another form of user interface such as could be provided by a mobile phone. Replacing a domain in this way would require changes to bridges from client domains, but not the domains themselves.

There are two sorts of bridges: explicit and implicit.

Explicit Bridges represent assumptions within the model of one domain concerning the existence of another domain that are explicitly represented as signals to and from external entities and the invocation of operations on such entities. In effect, the model of one domain assumes that some other anonymous domain, linked via an explicit bridge, will generate required signals, consume and act appropriately on signals it is sent and implement correctly any operations invoked on it. Explicit bridges are usually used to link application and intermediate abstraction domains to

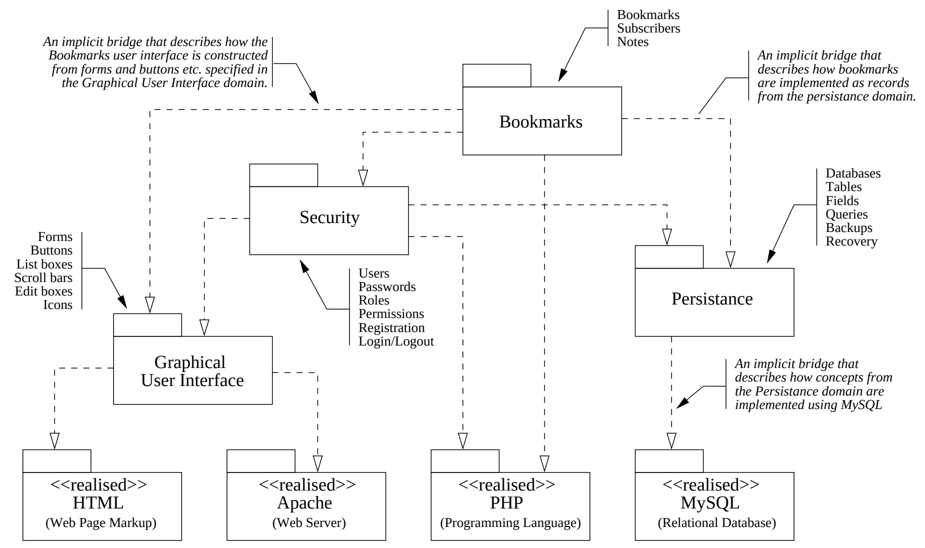

Bookmarks
Subscribers
Notes
An implicit bridge that describes how the Bookmarks user interface is constructed from forms and buttons etc. specified in the Graphical User Interface domain.
Bookmarks
An implicit bridge that describes how bookmarks are implemented as records from the persistance domain.
Databases
Tables
Fields
Queries
Backups
Recovery
Security
Forms
Buttons
List boxes
Scroll bars
Edit boxes
Icons
Users
Passwords
Roles
Permissions
Registration
Login/Logout
Persistance
Graphical User Interface
An implicit bridge that describes how concepts from the Persistance domain are implemented using MySQL
<<realised>> HTML (Web Page Markup)
<<realised>> Apache (Web Server)
<<realised>> PHP (Programming Language)
<<realised>> MySQL (Relational Database)

Figure 1: The Bookmarks Application Domain Chart with annotations describing the nature of each domain and bridge.

realised domains that represent external systems and legacy software.

In contrast to explicit bridges, *implicit bridges* represent assumptions within the model of one domain concerning the existence of another domain as a set of rules that direct the use of subject matter in the bridged domains to form elements of requirements specifications, design descriptions, implementation and other software engineering artifacts.

Implicit bridging rules can be categorised as *mapping*, *transformation*, or *weaving*. *Mapping* rules are used to state the correspondence between the subject matter of one domain and that of another domain. *Transformation* rules involve the *transforming* of subject matter of one domain into subject matter of another domain. *Weaving* rules are more complex and involve the principles of aspect orientation (Elrad, Filman & Bader 2002) where the subject matter of one domain is woven throughout the subject matter of another domain.

2.5 $^{X}_{T}UML$ and the software development lifecycle

$^{X}_{T}UML$ domains are identified, modeled and used throughout the software development lifecycle. During analysis, application domains, applicable intermediate abstraction domains and associated bridges are identified and modeled. During design, any other required intermediate abstraction domains are modeled along with all chosen implementation domains. Software architecture is captured in the form of translation rules associated with *implicit* bridges. During implementation, domains are systematically translated into deployable software in accordance with these architectural rules.

During maintenance, domains and bridges may be added, removed and/or modified to correct defects, add new functionality and to take advantage of new technology.

2.6 Model Compilation

One of the ultimate objectives of $^{X}_{T}UML$ is to rely on *model compilers* which are able to compile a set of domain models and associated bridges into artefacts such as executable programs that can run on a particular target software architecture. At present there are few model compilers available for educational purposes. The ones available mainly support simple architectures such as embedded C/C++ environments.

3 $^{X}_{T}UML$ and education

3.1 Our program requirements

We teach into two very different programs. Two of us lecture advanced Software Analysis and Design to undergraduate Information Technology, Information Systems, Computer Science and Software Engineering students. The other is responsible for graduate conversion programs in Information Technology which take students with degrees in the sciences and engineering and provide them with a mixture of programming, software engineering and general information technology courses. A feature of these graduate programs is that they have a central theme of virtual reality which is used as an orientation for a portion of their courseware.

In teaching into the general software engineering program, we aim to present the theory, practice and context of $^{X}_{T}UML$ with as much depth as time allows. We then aim to provide our students with a framework for applying $^{X}_{T}UML$ to their software engineering projects which acknowledges that they will not have access to a model compiler. A case study which reflects this approach is described in Section 4.

In teaching into the graduate program, we aim to integrate the Software Analysis and Design course with the students' initial foray into object oriented programming, design patterns and computer graphics programming. We acknowledge that these students will probably design and code at a lower level first but we demand that they then are able to evaluate what they have created within the $^{X}_{T}UML$ framework and that they are able to apply design patterns to restructure their software into *realised* domains which might be reused in the future. A case study which epitomises this approach is described in Section 5.

3.2 Benefits gained by adopting $^{X}_{T}UML$

The use of $^{X}_{T}UML$ in our programs directly addresses each of the problems associated with teaching UML and appropriate analysis and design methods outlined in the introduction.

1. ^X_TUML uses a small, well defined subset of the UML to represent domain models. It makes use of and clearly defines simple semantics for UML class diagrams, UML statechart diagrams and (derived) collaboration and sequence diagrams. To these are added an implementation of the UML action semantics (OMG 2003, section 2.16) which can be used to specify processing requirements within analysis models, and the UML Object Constraint Language (OCL) which can be used to specify constraints related to class and state chart diagrams.

   The use of such a profile, simplifies the teaching of UML while providing a notation sufficient to represent any model of domain data and behavior.

2. ^X_TUML provides a framework for the systematic development of software. A software system is decomposed into domains that each deal with a separate aspect of the system. Each of these domains is modelled using the ^X_TUML notation, action language and OCL. With the help of a suitable tool, these models can be verified as correct (meeting the requirements) at the analysis stage before any design or implementation decisions are made.

   This framework embodies a number of important software engineering principles and techniques we wish to introduce to students including an understanding of context, separation of concerns (between and within requirements, design and implementation), use of semi-formal information and behavioral specifications, interface specification, an emphasis on the early verification of requirements specifications, application independent design and large scale reuse of domain specifications and architectural design.

3. ^X_TUML models can be systematically translated into deployable software using tool support or by hand coding. This means that students can use a well defined method and UML profile to engineer specifications and designs, that can be translated onto popular implementation technologies such as PHP, Python and Zope even when such technologies are themselves poorly engineered.

As well as the above benefits, ^X_TUML provides students with a realistic approach to large scale reuse in the form of entire domain models and bridge descriptions.

3.3 Using ^X_TUML with limited tool support

While an ultimate objective for ^X_TUML is the use of model compilers, the reality is that commercial model compilers (there are no viable open source tools as yet) have the following disadvantages for us in our teaching programs and for our students when they move into their first jobs.

1. They are very expensive. (The model verifiers also cost money; but this is minor compared to the full model compilers.)

2. They are limited to simple software platforms such as embedded or command line C and C++ programs. Our students want to build systems using contemporary technology such as the Web, Java, .NET, PHP, Computer Graphics, Virtual Reality and so on. It is not only our students who want to build these systems. Industry also wants to move with popular implementation technology.

Our approach to these problems has been to try to teach *enough* ^X_TUML to give students *the good message* but then to emphasise the following points:

1. ^X_TUML modelling of domains is similar to the process that is carried out using other methods of object-oriented analysis. The differences are:

   (a) A single aspect of the software system is modelled within each domain. This dramatically simplifies the modeling exercise, but requires a mind shift that some students find difficult. With practice, however, average students eventually *see the light.*

   (b) The (subset) UML notation is strictly defined and the modelling detail is sufficient for the models to be executable. This strict subset of UML overcomes much of the ambiguity that students wrestle with regarding UML. Some of the fine detail from an ^X_TUML model can be lost but what remains are useful analysis models which can be turned into code (either by elaboration or translation). Keeping all the detail enables model verification at the analysis stage. Students are encouraged to maintain the detail and to verify their models whilst carrying out their software engineering projects.

2. Principles for the translation and weaving of ^X_TUML domains together can be stated in words and applied by hand by students in order to produce code. This can be turned into an iterative process: a first attempt at translation may produce code with short-comings, hence the process is to modify the translation (and/or the architectural) rules and try again. The advantage is that both a software product and a set of useful rules (design decisions) is developed which can then be applied to future products.

In effect, our approach to dealing with this limitation of existing tool support, has led to a further benefit of using ^X_TUML in our teaching program. It provides a framework within which students can capture and then systematically apply reusable design decisions.

4 Case Study I: Web based Bookmark Manager

Our first case study has already been introduced in the discussion of ^X_TUML above and is one of the examples we use in our undergraduate software analysis and design course. It comprises a simple web application that aims to demonstrate how ^X_TUML can be used to produce well engineered specifications and designs even when the target software platform is based on technology that supports few if any of the software engineering principles we deem important.

The Bookmarks application allows registered users to maintain a private set of web address bookmarks. The ^X_TUML domain chart for the application is depicted in Figure 1. Annotations on that figure indicate the nature of important domains and bridges.

The Bookmarks domain models the subject matter of bookmarks and subscribers to the system. Other concerns such as user interfaces, security and persistence, are the subject matter of other, separate, domains. Figure 2 depicts the class diagram for the Bookmarks domain.

The behavior of the Booksmark domain is specified using a state model as depicted in the State Chart

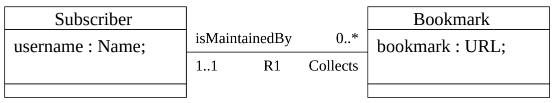


Figure 2: A Class Diagram that specifies the data requirements of the Bookmarks Domain

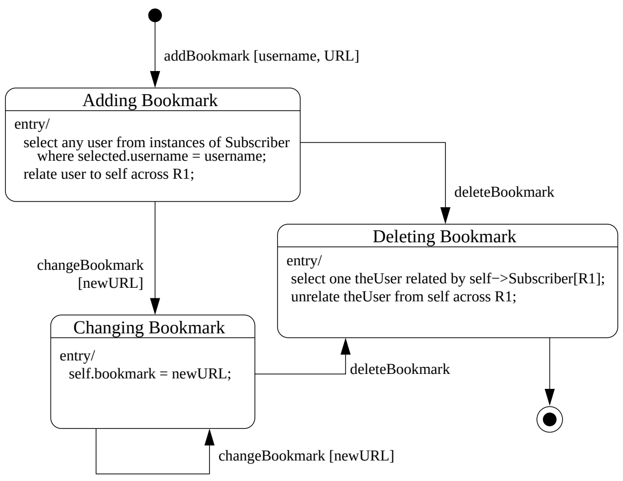

addBookmark [username, URL]
Adding Bookmark
entry/
select any user from instances of Subscriber
where selected.username = username;
relate user to self across R1;
deleteBookmark
changeBookmark
[newURL]
Deleting Bookmark
entry/
select one theUser related by self−>Subscriber[R1];
unrelate theUser from self across R1;
Changing Bookmark
entry/
self.bookmark = newURL;
deleteBookmark
changeBookmark [newURL]

Figure 3: A State Chart that specifies the required behavior of the Bookmark class.

of Figure 3. Note that the state model specifies the way in which instances of the bookmarks class are required to behave in response to events from outside the domain. The domain models do not specify or rely on the events coming from a particular domain.

The class diagram for the Security domain is depicted in Figure 4. Note that this domain model makes no reference whatsoever to aspects of the bookmarks or any other domain. The concept of a role, for example, is modeled in the security domain, but the actual roles required by the application we are building are not specified within the domain. It is the *implicit* bridge from the BookMarks domain to the Security domain that specifies the actual roles required. The security domain models how roles (whatever they are) relate to users and other parts of the security domain. The bridge will also specify what users with particular roles can do within the context of the Bookmarks domain (eg. what events they can generate and what data they can read and write).

The graphical user interface domain models concepts used to build GUIs including widgets, style guides and conventions. Bridges from the Security and Bookmarks domain are used to specify the layout of user interface screens and how their use impacts the operation of the Bookmarks and Security domains. This would include a description of what events are generated (for example, the *addBookmark[...], changeBookmark[...]* and *deleteBookmark* events depicted in Figure 3) by the user interface and

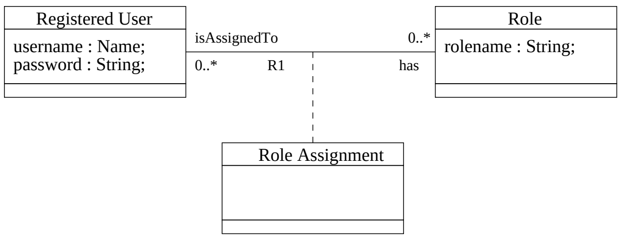
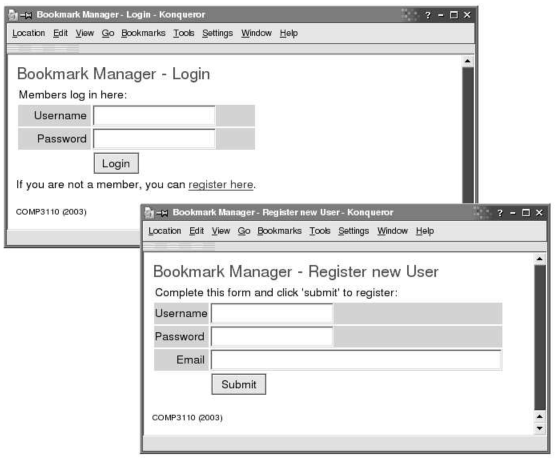


Figure 4: A Class Diagram showing data requirements of the Security Domain


Bookmark Manager - Login - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
Login
Bookmark Manager - Register new User - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
Submit

Bookmark Manager - Login

Members log in here:

Username

Password

If you are not a member, you can register here.

COMP3110 (2003)

Bookmark Manager - Register new User

Complete this form and click 'submit' to register:

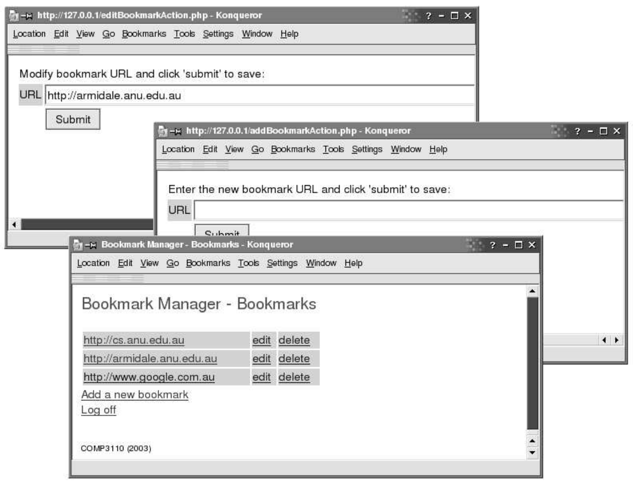
Username

Password

Email

COMP3110 (2003)

Figure 5: Screen shots of Security aspects of the User Interface


http://127.0.0.1/editBookmarkAction.php - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
Submit
http://127.0.0.1/addBookmarkAction.php - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
Submit
Bookmark Manager - Bookmarks - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help

Modify bookmark URL and click 'submit' to save:

URL http://armidale.anu.edu.au

Enter the new bookmark URL and click 'submit' to save:

URL

Bookmark Manager - Bookmarks

http://cs.anu.edu.au	edit	delete
http://armidale.anu.edu.au	edit	delete
http://www.google.com.au	edit	delete

Add a new bookmark

Log off

COMP3110 (2003)

Figure 6: Screen shots of Bookmarks aspects of the User Interface

what data is displayed and input via the user interface. Figures 5 and 6 depict some of the Security and Bookmarks web pages specified for our application.

Like the Security domain, the Persistence domain is modeled independently of Bookmarks and other domains. It models a way for data to be stored in a persistant fashion and how that data can be retrieved, backed up and recovered. It contains no reference whatsoever to details of the bookmarks, security or any other domain. The bridges from Security and Bookmarks to the persistence domain specify which data from these domains are to be persistent and the form they will take in terms of the concepts modeled in the persistence domain.

The domains at the bottom of the domain chart represent various aspects of the implemented software. They are all *realised* in that they are not modeled using $^{X}_{T}UML$ because they are generally well understood or already exist. GUI widgets will be implemented using HTML assembled into web pages that will be served by the Apache domain. PHP will be used to implement state actions associated with the Bookmarks and Security domain models. The MySQL domain will provide an implementation of the concepts modeled in the Persistence domain.

Once all of the *non-realised* domain have been modeled and verified, and all bridges have been specified, the student can systematically translate the model into artefacts of the HTML, Apache, PHP and

MySQL realised domains.

5 Case Study II: Draw3D

Our second case study is an example which we give to graduate students who are completing a software project in virtual-reality related areas. These students will have already taken courses which include material on object-oriented programming, design patterns, GUI construction and 2D and 3D computer graphics. They also have taken the course on Software Analysis and Design mentioned earlier.

Virtual Reality has been distinguished from many conventional software systems by the nature of the interface. At a minimum, a VR application will have a strong emphasis on 3D graphics and interactivity. Interactivity is mediated by a large variety of controls and sensors as well as by the graphical objects themselves. VR software must sometimes deal with complicated sets of asynchronous events whilst taking time into account in a similar way to real time systems. The applications themselves are often derived from simulations (of aspects of the real world) interwoven with narratives and game logic.

On a practical level, VR software development is a specialist activity because

- a deep knowledge of 3D graphics APIs, and graphics modelling software, is needed to produce a quality experience with novel behaviors
- the interface involves novel, and changing, controllers and sensors
- the hardware display platforms can vary greatly
- human computer interface design for VR systems is still an active area of research rather than routine application.

A common problem we notice with our graduate students is that they become obsessed with the details of working with the particular scene graph API (we use Java3D in our computer graphics course). When the scene graph *takes over* a program, the game logic becomes encapsulated in particular *behavior nodes* which are interspersed with geometrical information in the scene-graph tree. It is very difficult to see how these behaviors knit together to produce the interactive dynamics of the application. Such a structure is very difficult to extend and modify. Our hope has been that the separation of concerns that students have learnt in their $^{X}_{T}UML$ course can be brought to bear on this type of software system.

This particular case study is an interactive, 3D-line-drawing application. An initial message is displayed which invites participants to press a button to obtain the main program menu. After choosing *New Line* for the first time, the main menu disappears and is replaced by a dancing cursor which can be moved in each of the 6 coordinate directions. Pushing a *pen down* button places a *blob* of randomly coloured ink onto the screen. Subsequent movements of the cursor draw a 3D line segment which can be moved in any direction until the second end-point is defined. Participants can build up a picture which is made up of several, randomly coloured, lines each of which can have many line segments and blank segments.

The graduate students generally do not possess a software engineering or IT background and so, many of them, whilst finding the concepts of $^{X}_{T}UML$ relatively easy to understand, are unable to realise the full meaning and purpose of domains and bridges - especially *implicit* bridges. Hence the approach that has been taken with respect to the Draw3D project more closely aligns with familiar concepts such as subsystems and explicit bridging even though the domain chart in Figure 7 clearly supports separation of concerns. Explicit bridging seems to be easier for students to deal with when the application is strongly oriented to the user-interface, especially where there are obvious domain counterparts between the Draw3D domain and the Models, Scene Graph and GUI domains. Also, under normal circumstances specific, realised implementation domains, like those shown in the lower part of Figure 7, would not be seen on a domain chart during the early stages of development. However, part of the purpose of the Draw3D project is to develop in Java and to that extent such realised domains represent an implementation constraint.

As for Case Study I, each domain is modeled using the strict rules of $^{X}_{T}UML$ forming firstly, class diagrams for each domain and then modeling any behaviour of classes using state charts. The Draw3D domain deals with the concerns of importing, editing and creating 3D graphical models, whilst the Scene Graph domain is concerned with appearance and connectivity of the graphical models described in the Draw3D domain. The GUI domain is concerned with the interface to manipulate (import, edit and create) the graphic entities concerning the Draw3D domain. The explicit bridges between the domains generally describe which events within a client domain need to be transmitted to a server domain, providing an asynchronous connection between the domains. A more strongly coupled connection is possible if specific methods within a server domain are activated from classes within a client domain. These concepts, whilst counteracting the ideals of separation of concerns, do begin to steer the graduate students toward a better understanding of the ideal where domains can be treated in total isolation and be reused as required within other applications.

6 Further Work

Our third year Information Technology and Software Engineering students are required to complete a year long group project. As part of this effort, they are required to produce a series of documents including a Software Requirements Specification (SRS) and a Software Design Description (SDD). This year, most of the groups are using $^{X}_{T}UML$ to help in the construction of their SRSs and more than half of the groups are continuing with $^{X}_{T}UML$ in their designs. Early indications are that students have difficulties in the following areas:

1. breaking away from an implementation focus
2. separating concerns within the SRS
3. mapping $^{X}_{T}UML$ work products to IEEE software documentation standards and guidelines

Despite these difficulties, it appears that many students have gained confidence in using $^{X}_{T}UML$ and that they learnt a great deal about how to develop an effective SRS and SDD.

At the conclusion of the projects, we plan to evaluate the effectiveness of using $^{X}_{T}UML$ in student projects.

7 Conclusion

We have been incorporating the teaching of $^{X}_{T}UML$ into our various courses since 2001. The case studies described here illustrate that the method has much

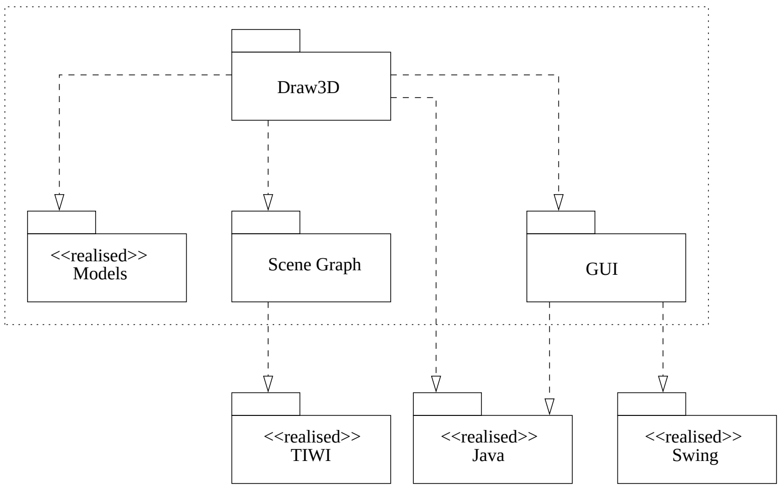


Figure 7: Domain Chart for the Draw3D program

to offer software development in a wide range of application areas.

Taken to its extreme, and with the support of appropriate tools and model compilers, $^{X}_{T}UML$ raises the level of abstraction beyond that of UML models. Some people would argue that this future state is with us now and that this is the way that software development should proceed. Our conclusion is that there are clear benefits in incorporating the major elements of $^{X}_{T}UML$ into a teaching program even if the full model-compiler route is not carried out. These benefits are

1. The use of a strict subset of UML which is easier to understand than the full notation and which does not suffer from ambiguity or misuse.

2. A strong requirements-focus in software development. It is possible to verify that the requirements are being met at the analysis stage.

3. A strong emphasis on a separation of concerns. This separation starts at the analysis stage and is carried through the translation process. Students are encouraged to focus on the nature of concerns such as security, persistence and graphical representation without being trapped into coding in the particular way of an API. (This has been of great benefit to the virtual reality application in the second case study.)

References

Booch, G. (1994), *Object Oriented Design with Applications*, Addison-Wesley. ISBN 0805353402.

Elrad, T., Filman, R. & Bader, A. (2002), ‘Aspect-oriented programming:introduction’, *Communications of the ACM* **44**(10), 29–32.

IEEE (1998), *IEEE-STD-830, Recommended Practice for Software Requirements Specifications*, IEEE, New York, NY.

Kennedy Carter (2003), *iUML executable UML modeling tool*, Kennedy Carter Ltd., Surrey, UK. Details available from http://www.kc.com (last accessed 3 November 2003).

Mellor, S. & Balcer, M. (2002), *Executable UML, A foundation for Model-Driven Architecture*, Addison-Wesley, Indianapolis, IN.

OMG (2003), *Unified Modeling Language Specification, version 1.5*, Object Management Group, Needham, MA. Available from http://www.omg.org (last accessed 3 November 2003).

Project Technology (2003), *BridgePoint Development Suite*, Project Technology Inc., Tucson, AZ. Details available from http://www.projtech.com (last accessed 3 November 2003).

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorenson, W. (1990), *Object Oriented Modeling and Design*, Prentice Hall. ISBN 0136298419.

Shlaer, S. & Mellor, S. (1992), *Object Lifecycles, modeling the world in states*, Prentice-Hall, Upper Saddle River, NJ.

Starr, L. (2001), *Executable UML, A case study*, Model Integration, LLC. Available from http://www.modelint.com (last accessed 3 November 2003).

Starr, L. (2002), *Executable UML, How to build class models*, Prentice-Hall, Upper Saddle River, NJ.